AFRL-IF-WP-TR-2002-1522

TEST GENERATION FOR VERY HIGH-LEVEL DESIGN LANGUAGE (VHDL) SPECIFICATIONS USED IN AVIONICS

Dr. Mohamed F. Chouikha

Howard University
College of Engineering, Architecture and Computer Sciences
2300 Sixth Street NW
Washington, DC 20001-2323

AUGUST 2002

Final Report for 27 June 1997 – 10 May 1999

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

NOTICE

Darrell Barker

DARRELL BARKER
Project Engineer
Embedded Info Sys Engineering Branch
Information Technology Division

[illegible]

JAMES S. WILLIAMSON, Chief
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE ***(DD-MM-YY)*** August 2002	**2. REPORT TYPE** Final	**3. DATES COVERED** ***(From - To)*** 06/27/1997 – 05/10/1999

4. TITLE AND SUBTITLE TEST GENERATION FOR VERY HIGH-LEVEL DESIGN LANGUAGE (VHDL) SPECIFICATIONS USED IN AVIONICS	**5a. CONTRACT NUMBER** F33615-97-C-1127
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62204F
6. AUTHOR(S) Dr. Mohamed F. Chouikha	**5d. PROJECT NUMBER** 6096
	5e. TASK NUMBER 40
	5f. WORK UNIT NUMBER 38
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Howard University, College of Engineering, Architecture and Computer Sciences, 2300 Sixth Street NW, Washington, DC 20001-2323	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Information Directorate, Air Force Research Laboratory, Air Force Materiel Command, Wright-Patterson Air Force Base, OH 45433-7334	**10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/IFTA
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S) AFRL-IF-WP-TR-2002-1522

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT ***(Maximum 200 Words)***

This report describes a technique that automatically generates test pattern sequences for digital systems based on VHDL inputs. This research aims to design a method for detection and elimination of inconsistencies in the Extended Finite State Machine (EFSM) directed graph model of the VHDL by splitting the graph vertices and duplicating the edges minimally. It focuses mainly on detection and removal of condition-to-condition and action-to-action inconsistencies. Section 1 describes the general problem and lists different tools that already exist. Section 2 describes the algorithm for detection and removal of condition-to-condition inconsistencies. Section 3 presents a pseudocode of the combined detection/removal condition-to-condition inconsistencies. Section 4 describes the algorithm for detection and removal of action-to-action inconsistencies. Section 5 presents a pseudocode of the combined detection/removal action-to-action inconsistencies. Section 6 describes the implementation of the algorithm in C along with some examples.

15. SUBJECT TERMS
test patterns, test generation, electronic computer-aided design, VHDL

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** (Monitor)
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	SAR	50	Darrell Barker **19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 255-6548 x3605

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 Introduction

As the avionics industry progresses in providing new and enhanced capabilities, the number of products which must interoperate with one another is growing at a dramatic rate. In order for such complex heterogeneous systems to function properly, formal specification and testing methods are required. In addition, the formal methods utilized in specification and testing stages will significantly accelerate the development cycle, which in turn will reduce the development costs and at the same time improve the quality by providing comprehensive tests.

Increasing complexity of hardware necessitates the use of high-level languages, such as VHDL, in the specifications. The methodologies presented in this descoped proposal will be developed primarily for systems that are defined in VHDL. However, the adaptation of these methodologies to the system (i.e., functional) level testing of VLSI designs will be straightforward. Automated test pattern generation (ATPG) from high-level circuit descriptions hence becomes a critical problem in VLSI circuit design and testing. Although the automated test pattern generation is NP-complete for the general case, efficient methods are essential to reduce the time complexity to generate tests and to nearly minimize the number of tests.

This report addresses the first steps in the design of automated test pattern generation based on VHDL specifications. Gate-level [1] and finite state machine (FSM) models [2][8] cannot address the complexity offered by the high-level description languages such as VHDL. Approaches based on extended FSM (EFSM) models [3][10] are not applicable to most real-life circuits, since they consider test generation on a single path at a time in a high level circuit description (note that the number of paths is in the order of 1010 or larger in typical medium size specifications).

This program aims to extend the results from test minimization techniques reported in the communications protocol testing field to reduce the number of tests [4]. Currently, these techniques are limited to FSMs and not applicable to VHDL descriptions in general. Furthermore, this descoped proposal will explore the symbolic execution technique [5], which is extensively used in the test generation of general software, to resolve the conflicts and inconsistencies among conditions and actions of a VHDL specification. Symbolic execution is not directly applicable to VHDL descriptions since only one path is tested at a time, which requires unacceptably large analysis time.

Extended FSM (EFSM) models which include the variables used in the state transitions of a state machine are more suitable to represent complex circuits. The exponential growth in the number of paths (versus the EFSM size) in a FSM specification combined with the inconsistencies among the conditions and actions make the test generation problem NP-complete. A test generation method for EFSMs is reported to handle path explosion problem [3]; however, it cannot handle action-to-condition and action-to-action inconsistencies. There are several techniques that address such inconsistencies in the test generation. Symbolic execution technique defined in software engineering has been applied to RTL descriptions and microprocessor circuits [9]. Equivalent techniques have been proposed for functional level descriptions in [10]. The main advantage of these techniques is their ability to model real-life, complex hardware specifications. But, unfortunately, they lack the efficiency in test generation time and the number of tests that they generate. This limitation is due to the path-oriented nature of these approaches. In other words, they consider tests for a single path at a time in the specification. Since the number of paths can be in the order of 1010 or larger, their applicability to real-life specifications is severely impaired.

The proposed approach will utilize an EFSM model to represent the VHDL specification, therefore will not have the shortcomings of the simplistic FSM-model approaches [6]. In addition, the proposed approach will handle the path explosion problem of [8] by using symbolic execution techniques to limit the number of split nodes in the EFSM graph model.

The objective is the extension of the symbolic execution theories to handle VHDL specifications. This will include the development of a set of algorithms for detection and elimination of inconsistencies in the EFSM directed graph model of the VHDL specification by splitting the graph vertices and duplicating the edges minimally. Even for specifications of limited size, various inconsistencies may occur among the conditions and actions during the test generation : condition-to-condition, action-toaction inconsistencies. Existing research results of the authors presented the necessary and sufficient conditions for these inconsistencies [12]. The resolution of such inconsistencies further complicates the test generation procedure as described below:

1. Condition-to-condition inconsistency: An input set that satisfies a condition in the specification can be inconsistent with another condition appearing further in the specification.

II. Action-to-action inconsistency: If actions use variables that are modified by other actions appearing above it in the specification, the values of such variables may depend on resolving inconsistencies between the conditions of this statement and the conditions in the above statements.

Effective methods are needed to resolve various inconsistencies that can occur during the analysis of a path in a VHDL specification. Furthermore, minimization techniques are crucial to reduce the number of tests that are generated based on analyzing unacceptably large number of paths 1010 or larger in the specification.

Test generation of EFSMs has been an active research area in software testing since software programs can easily be modeled as EFSMS. Symbolic execution technique is the most popular approach in software engineering to generate tests, prove correctness of programs and reduce the program size [5].

Regular execution of a program consists of running the program with input data and generating a set of outputs and/or updating some of the variables (if any). Depending on the input data, a path of the program is exercised. In symbolic execution of a program, however, for a given path, the input data values are replaced by symbolic values. The results are a set of expressions of inputs and constants. In other words, symbolic execution does not directly execute a program; it generates algebraic, rather than numeric, values.

In symbolic execution, a program is represented as a data flow graph where the decision points (i.e., conditional statements) are the nodes, and the assignment statements are associated with the edges between decision points. A path from an entry point to an exit point of the program corresponds to a set of inputs, condition predicates, and assignment statements. A symbol, rather than a value, is assigned to each input. Symbolic execution traverses a path in the program while evaluating each assignment and condition in terms of input symbols and constants. Therefore, after a path is traversed, the output variables will be represented in terms of symbols and constants. For the output to be meaningful, the conditions encountered during the path traversal must be feasible (i.e., possess a solution).

In general, test generation by using symbolic execution technique proceeds as follows:

Step 1. Select a path in the data flow graph.
Step 2. Evaluate variables in terms of input symbols along the path and conditions.
Step 3. Check if the path conditions are feasible to obtain the test case. If the path conditions obtained in Step 2 are feasible, the solution to the compound path conditions gives the input values to execute the path which constitutes a test for this path.

For general software, symbolic execution techniques have weaknesses to address complex loop functions, uninitialized arrays, etc. However, we believe the VHDL specifications are simpler than general software, and the shortcomings of symbolic execution are not applicable to hardware description languages. Thus, the major issue to be addressed in the use of symbolic execution techniques for hardware testing is solely the explosion of the number of generated tests. Since a single path is considered at a time in path-based testing methods, the number of paths in a hardware specification and hence the number of tests can become prohibitively large.

The approach starts with a VHDL specification. After conversion of the VHDL specification to the EFSM model and using the symbolic execution techniques, the EFSM model will be converted into an FSM model by splitting the nodes in the EFSM graph. The total number of vertices in this FSM graph is expected to be reasonably bounded (i.e. not explosive).

This report considers the problem of detection and removal of Condition-to-condition and action-to action inconsistencies. In summary, the technically challenging parts of the proposed method are the conversion of EFSM to the FSM and the generation of the test cases.
First, an algorithm is developed to detect the inconsistencies among the conditional statements of the EFSM graph. Then, a second algorithm removes the inconsistencies from the EFSM graph by splitting the vertices. And, both algorithms are implemented in C. The above steps are repeated for the case of the action-to-action inconsistencies.

To enhance the implementation two tools namely V2S[F and SAVANT were used. A list of state-of-the-art tools from the perspective of automated test sequence (or test vector) generation directly from the VHDL specifications is provided below. This list is based on the material provided by the tool vendors in the World Wide Web.

Tool Name: SPW Finite State Machine Editor
Provider: Cadance Design Systems
Characteristics: This is a finite-state machine (FSM) editor that enables the user to automatically generate VHDL code (and other languages) from state diagrams. This tool is mainly designed for the state diagrams that are either Moore or Mealy machines. However, the tool description also mentions that the FSMs may have boolean .--d expressions for a given state. Therefore, we assume that the EFSM models are also within the scope of the tool. State machine is described in the so-called Action Description Language (ADL), which is a C-like language. The tool provides the user with the capability of defining concurrent FSMS. This way, a user can suppress the number of states into a much smaller size (where possible) by modeling the system as communicating FSMs as opposed to a single FSM. The tool has the capability of "on-line verification of FSM action specifications (states and transitions)." This is the ability to check the consistency among the various state transition conditions and the actions. It appears that this ability is a somehow extended compiler syntactical checker designed specifically for the ADL language.

Tool Name: StateCAD
Provider: Visual Software Solutions
Characteristics: This tool translates the FSMs to VHDL (and other languages) This tool is also sold by Viewlogic (Synopsis) and Synario (Minc) under their own labels. The FSM (Mealy or Moore machines) are specified by using the so-called "drawing engine" which provides a graphical tool to draw the FSM model. Concurrent FSMs are allowed to be specified in the graphical drawing tool (including both the synchronous and asynchronous transitions). After the FSM is specified, StateSIM lets the user analyze the behavior of an FSM specification in a graphically animated format. It allows the user to initialize the model into any state and simulate the behavior starting from that state. StateSIM also checks for the consistent specification of inputs and outputs within the entire model. The tool automatically finds more than 200 "inconsistencies" such as stuckat faults, redundant outputs, conflicting outputs, and unused outputs, vector range violations, reserved word usage. The advertisement claims that StateCAD "exhaustively analyzes" the FSM model to locate these problems. Logic Wizard automatically creates the data flow structures for registers, latches, counters, multiplexers, etc.

Tool Name: ViewFSM
Provider: Viewlogic
Characteristics: ViewFSM optimizes, and architecturally maps the design. The input can either be VHDL or a state machine created by using the state diagram graphical editor of the tool. If the graphical editor is used, VHDL code is generated by the tool. After the specification is completed, ViewFSM optimizes the code by checking for unreachable states, equivalent states. It also architecturally maps the code by specifying the direct outputs, removing unused code, and specifying the state machine encoding. This architectural mapping generates RTL-level VHDL. Although it is not clear from the advertisement how it is done,

ViewFSm claims to automatically create tests and simulate the design behavior by using these tests.

Tool Name: Visual HDL for VHDL
Provider: Summit Design
Characteristics: The tool accepts the state machine designs in graphical format. It can also import VHDL and generate its graphical representation based on the HDL source code. The simulation and debug environment of the tool allows for the selection of a given path in the code and its simulation. As an option, the tool provides Text-to-State Machine options which converts VHDL descriptions into graphical state diagrams. Visual Testbench enables regression testing environment (recapturing the data to test the design for later versions, etc.)

Tool Name: Statemate MAGNUM
Provider: i-Logic
Characteristics: This tool represents a specification by a graphical language. It is also possible to specify a system by using C or Ada languages. Various types of verification for the specifications are available (similar to the consistency checking in StateCAD, etc.). Trailblazer(TM) simulator provides graphical animation of the specification (i.e., simulation). Interactive debugging of the code is possible by using such a simulator. The tool provides interfaces to various other software development tools and environments including VAPS, Matrix AutoCode, RTM, and DOORS.

2.0 Condition-to-Condition Detection and Removal (CCDR) Algorithm

It's assumed that EFSM model of the VHDL specification can be obtained from the Data Flow Graph (DFG) derived from a tool such as V2SIF or SAVANT. The following sections describe the algorithms for detection and removal of condition-to-condition inconsistencies of this EFSM. Although these two processes are described separately, they are combined in the developed pseudo-code and its C-code implementation.

In the EFSM, a node structure is defined by its number of outgoing edges, the condition variable, operation, scalar value, and the destination node of each outgoing edge. The root node is defined as the initial node of the DFG of the EFSM model of VHDL specification. The following array triplets define a path leading to a node j of the DFG:

C[I][p][e][v] : the coefficient for the accumulated conditions ***along path p to node*** t ***where,***

i *: node number (index) = 0 to N- 1; N is the total number of nodes in the DFG.*

p *: path number (index) [of one of the P paths leading to node* **I***] = 0 to P- 1.*
e *: edge number (index) [of an edge, length, in path* **p** *leading to node* **i***] = 0 to E-1; E is the number of edges in path p.*

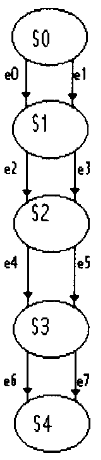
v *: variable index [of edge* **e** *of path* **p** *leading to node* **I**] *= 0 to V-1; V is the number of variables in DFG.*
C[I][p][e][v] = 1*,* if the edge (condition) exist

0, otherwise.

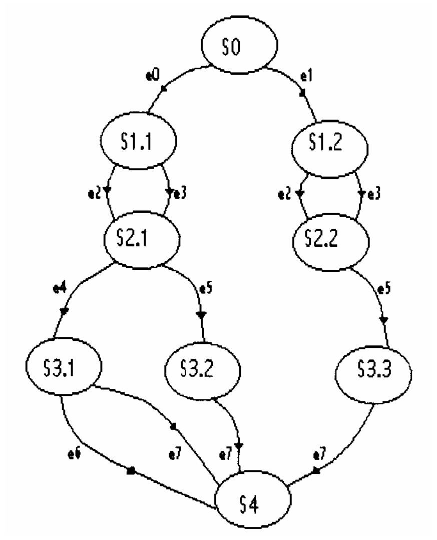
OP[I][p][e] = ***o*** **:** operators for each accumulated conditions *(of edge* **e)** *of path* **p** *leading* ***to*** *node* **i** which can be either >, <, >=,== **<=,** or !=. *The number of OP for each path* **p** *is E,- .ie., for each path* **p** we *have* **OP[I][p][e=0,1,…E-1].**

D[I][p]]e] = ***d*****:** scalar values for the accumulated conditions of edges of path m leading to node n. *The number of* **D** *for each path* **p** *is E; .ie., for each path* **p** we *have* **D[i][p][e=0,1,…E-l].**

Example. The DFG of an EFSM example, before and after the removal of condition-to-condition inconsistencies, are shown in Figure 1. Nodes in this example are indexed *(n):S0 = 0, S.1.1 = 1, S1.2=2, S2.1=3, S2.2=4, S3.1=5, S3.2=6, S3.3=7, and S4=&* Edges are numbered here in the order they appear in the DFG. See Table I for the definition of conditions. For each node, however, they are indexed accordingly. For instance, e2, e3 are indexed as outgoing edges 0 and I for nodes S1.1, S1.2, as are edges e4 and e5 for node S3.1 There are two variables in the graph, *X and Y, indexed by v* = 0, 1, respectively.



a) EFSM (with C-C inconsistencies)



b) EFSM (with inconsistencies removed)

Figure 1: DFG of an EFSM Example.

TABLE 1. Conditions

Edge	Condition
e_0	$x > 10$
e_1	$x \leq 10$
e_2	$y > 0$
e_3	$y \leq 0$
e_4	$x > 20$
e_5	$x \leq 20$
e_6	$x > 30$
e_7	$x \leq 30$

Consider node S3.3 (n=7) for instance; there are two paths leading to this node: m=0 (el, e2, e5) and m=l (el, e3, e5). The triplets for this node then should be:

For m = 0,
C [7][0][0][0] = 1
C [7][0][0][1] = 0
C [7][0][1][0] = 0

For m= 1,
C [7][1][0][0] = 1
C [7][1][0][1] = 0
C [7][1][1][0] = 0

C [7][0][1][1] = 1	**C** [7][1][1][1] = 1
C [7][0][2][0] = 1	**C** [7][1][2][0] = 1
C [7][0][2][1] = 0	**C** [7][1][2][1] = 0
OP [7][0][0] = "<="	**OP**[7][1][0] = "<="
OP [7][0][1] = ">"	**OP** [7][1][1] = "<="
OP [7][0][2] = "<="	**OP** [7][1][2] = "<="
D [7][0][0] = 10	**D** [7][1][0] = 10
D [7][0][1] = 0	**D** [7][1][1] = 0
D [7][0][2] = 20	**D** [7][1][2] = 20

Furthermore, the above imply:

for m = 0,

```
|1 0| | X | | <= | | 10 |          X <= 10
|0 1| | Y | | >  | |  0  |          X > 0
|1 0| | X | | <= | | 20 |          X <= 20
```

and for m = I

```
|1 0| | X | | <= | | 10 |          X <= 10
|0 1| | Y | | <= | |  0  |          X <= 0
|1 0| | X | | <= | | 20 |          X <= 20
```

2.1 Algorithm for C-C Inconsistencies Detection (CCD)

To detect a condition-to-condition inconsistency, the array triplets of each node j are checked with all of its outgoing edges. If any edge in the triplets has a variable that matches a variable of the outgoing edges, their operations (conditions) and scalars are checked for inconsistent conditions; if one detected, the node i (above node j) that has the inconsistency causing edge is marked for the removal of the inconsistency.

2.2 Algorithm for C-C Inconsistencies Removal (CCR)

To remove an inconsistent path, the destination node of the inconsistency causing edge of node i is split (duplicated) by the number of incoming edges of node i. Splitting propagates in depth down to and including node j, duplicating intermediate nodes, accordingly. All array triplets of paths leading to node j are once again compared with its outgoing edges. Inconsistent paths are removed by deleting their array triplets from the *set*. ***Example:*** Consider the EFSM shown in Figure la. Starting from node 0 and in an indepth manner along path el ➔ e2, a C-C inconsistency is detected for outgoing edge e4 of node S2 (X=<O and X >=20). Therefore, node SI is first split by 2, and node S2 is split by 2, accordingly. Checking for inconsistent paths, results in the removal of outgoing edge e4 from node S2.2 (a split copy of node S2).

The algorithm starts from node 0, the root, and moves in an in-depth-first manner to detect and remove inconsistencies. When it reaches either a leaf node or a previously visited node (in the current in-depth path), it back tracks to the root node and starts an indepth pass for the next outgoing edge of the root node. The algorithm terminates when all outgoing edges of the root node are exhausted.

3.0 CCDR Pseudo-code

In coding of the described CCDR algorithm, the detection and removal processes are combined. The developed pseudo-code is given in Appendix A. It mainly consists of a harness part, called DFM - DRIVER(root), which traverses the DFG in a depth-first manner and activates a detection-removal function, DET_REM_COND_COND(cur, next, mode) for each selected node of DFG. There is also a back track function, called BACK_TRACK(cur, outdeg, DFM_DONE) that when called, moves back one length along the current path, retrieves the current node, and sets for the next outgoing edge of the current node (at this length) for further C-C detection and removal. DET_REM_COND_COND sets a flag, mode, to PUSHING or POPPING, indicating indepth pass or back-track request. Similarly, BACK_TRACK uses a DFM_DONE flag to indicate if there's no more outgoing edge from the root node.

The DFM_DRIVERO starts a point (A); while DFM_DONE=0, it determines the next node (destination) of the outgoing edge (initialized to 0) of the current node (initialized to root), and calls DET - REM - COND - COND function. Followed at point (B); if still PUSHING, it saves the current node and its outgoing edge, updates the path_nodes[length] and path_ outdeg[length] arrays, increments the length and in-depth to the next node for detection and removal; if POPPING, it moves horizontally to the next outgoing edge and calls for BACK_TRACK.

The BACK_TRACK starts at point (C) by testing whether that outgoing edges at the current node are exhausted; if so, it back tracks one length and tries the next outgoing edge at the new level. It sets DFM_DONE to 1, when all outgoing edges of the root node are exhausted.

The DET_REM_COND_COND starts at point (D) by first extends the current path along to all outgoing edges of the current node i to the next node (destination node selected by DFM_DRIVER), by constructing new array triplets. If the next node is either a leaf node or a previously visited node (no new triplet added to the existing triplets of the next node), mode is set to POPPING and returns for backtracking. Otherwise, mode is set to PUSHING and proceeds to inconsistency detection (CCD) as described above. The CCR algorithm described above is then executed to split the appropriate nodes and to remove inconsistent paths (invalid triplets).

4.0 Action-to Action Detection and Removal (AADR) Algorithm

In what follows, the A-A inconsistencies detection and removal algorithms are introduced along with general and VHDL examples for completeness.

4.1 Algorithm for A-A Inconsistencies Detection (AAD)

A. Problem Statement

A1. Action-to-Action Inconsistency Definition:

In an EFSM representing a VHDL specification, consider an action that modifies a variable (say vl) by using other variable(s) (say v2) and/or constants. If v2 has multiple values due to previous modifications in the specification, an action-to-action inconsistency (AAI) occurs. In general, whenever a variable used in an action has more than one value (due to previous modifications) an inconsistency is present since it is not clear which value of the variable will be used at that point.

A2. Examples for the general case:

Consider the following action where x is assigned two different constant values based on cond1:

```
if (condl = 1) then

        x= 10;

else

         x = 20;

endif
```

If x is used later in the specification, an AAI will occur since x can have two different values depending on cond1 value.

Consider the following action where y is updated differently based on cond1:

```
If (condl) then

         y=y+ 11;


else
```

```
            y = y + 21;

        endif
```

Regardless of the initial value of y, an AAI will occur if y is used later in the specification since y will have two different values.

A3. Examples for the VHDL case:

The following portion of a VHDL specification contains the AAI based on assigning constants to variables or updating variables in actions differently:

```
if (condl = 1) then

        x = 10;

        y = y + 11;

else

         x = 20;

         y = y + 21;

endif
........

if (cond2 = 1) then

        z= 1;

else

         z = -1;

endif

........

if (cond3 = 1)

         x = x + y + z;

end
```

In the above example, the if statement with cond3 uses x, y, and z in its action where each has different values based on which conditions are satisfied above (cond1 and cond2). If, for example, (cond1 = 1) and (cond2 = 1) were satisfied above, the new value for x in if (cond3=1) will be:

$$x = (10) + (y + 11) + (1) = y + 22$$

On the other hand, if (cond1 = 0) and (cond2 = 0) above, the new x value in if (cond3 = 1) will be:

$$x = (20) + (y + 21) + (-1) = y + 40$$

Based on the initial value of y, the following values are possible for x after if(cond3=1)else is executed:

$x = y + 40$	(condl=l, cond2=1, cond3=1)
$x = y + 22$	(condl=0, cond2=0, cond3=1)
$x = y + 20$	(condl=l, cond2=0, cond3=1)
$x = y + 42$	(condl=0, cond2=1, cond3=1)
$x = 10$	(condl=l, cond2=1, cond3=0)
$x = 20$	(condl=0, cond2=0, cond3=0)
$x= 10$	(condl=l, cond2=0, cond3=0)
$x = 20$	(condl=0, cond2=1, cond3=0)

Similarly, y and z has 2 possible values each, based on the values of condl and cond2.

Since only the AAI's are considered at this point in the project, it is assumed that the conditions of cond1, cond2 and cond3 are conflict-free and that they are using variables other than x, y, and z.

A4. Action-to-Action Inconsistency Detection (AAID) Method:

The proposed method traverses the directed graph representation of the EFSM in a breadth-first manner. The actions of the conditional statements are examined and the variable updates are accumulated. When different values for a given variable on the right-hand side of an action are observed, AAI is detected.

B. Proposed Solution

Bl. Solution Formulation

In the following algorithms, only the inconsistencies among the actions of VHDL specification statements are considered. In a breadth-first-manner, all actions are passed down in the directed graph G representing the EFSM model of the VHDL specification. In this process, the right hand sides of the updated variables are checked in each action statement. If a variable x is updated by another variable y, where y is modified differently in two or more parallel edges above, an action-to-action inconsistency is detected (x and y may or may not be the same variable).

In the algorithms, each node is associated with a pair of action matrices:A[N][Q][V][V]: action update matrices representing all the modifications applied to every variable. The upper bounds for N, Q, and V are:

N: number of nodes in G representing EFSM

Q: number of valid paths accumulated up to a given node

V: number of variables in G representing EFSM

B[N][Q][V][1]: scalar values for the modified variables.A[root][0][V][V], which is defined for the initial node of the EFSM graph, is initialized to be the identity matrix of I[V][V]. Elements of B[root][0][V][1] are initialized to 0.

For simplicity, all variables are assumed to be homogenized for all actions. The converter from VHDL to the EFSM should handle this assumption by placing coefficients of 0 for the variables that are not used in a given action. The actions and the conditions in the G of the EFSM are assumed to be in linear form.

In a given node n - i in G, an outgoing edge e-i may have several actions. Each action is in the form of:

X_r = D[I] [V] (matrix mult) X[V] [I] + s_r where the V-element vector D represents the coefficients of the variables which modify x_r, X is the vector of variables, and s-r is the scalar used in this action.

B2. AAID Algorithm

The graph G representing the specification is traversed in an breadth-first manner.
During the traversal, at each node, for all edges leaving the node, the AAI are checked by comparing the matrix pairs associated with the node and the actions in the edge. When an AAI is detected, it should be removed from the graph by splitting the nodes (not provided at this point). Afterwards, the action matrix pairs are modified by using the actions on the edge.

A high-level description of the AAID algorithm is as follows:

```
in a breadth-first manner visit each node i:
/* check inconsistencies for edges leaving node i */
for (each ending node j from node i) {
    for (each outgoing edge from node i to a node j) {
        for (each action on an edge) {
                check for AAI by comparing the accumulated matrix pairs for node i
                with the actions on the edge.  If the edge actions are using a variable
                updated differently above, AAI is detected.
                }
 }
for (each parallel edge from node i to node j) {
        Based on the actions on the edge, update the action matrix pairs so that they
         reflect the latest action modifications on the variables.
         }
}
```

The breadth-first traversal does not continue when reaching either the root node, or a node that is visited 3 times (an arbitrary number). This limitation is needed since the condition-to-condition inconsistencies are not considered in the graph (which will cause infinite-loops since without conditions loops do not end). When AAID is combined with the condition-to-condition inconsistencies, this limitation will be removed (See also Section B7 on the limitations of the AAID algorithm).

B3. AAID Algorithm compared to methods in literature

AAID uses an approach similar to the symbolic execution method defined in the field of software engineering. The symbolic execution is designed for the data flow graph representation of a program whereas AAID is defined for the EFSM model. In general, EFSM model contains more relevant information from the test generation point of view than the data flow graph. This extra information of EFSM is especially important when eliminating the inconsistencies.

B4. AAID Applied to the General Case

The application of AAID for the general case is the same as in the VHDL case (See

Section B5).

B5. AAID Applied to the VHDL Case

In general, the actions in each edge will be in the forin of.

x a00 a01 a02 x b00

y = al0 al1 al2 x y + b0l

z a20 a2l a22 z b02

For example: y = y + 11 will be:

x 0 0 0 x 0

y = 0 1 0 x y + 11

z 0 0 0 z 0

The EFSM graph of the VHDL specification from Section A3 has 4 nodes (0 through 3) and 8 edges (2 edges leaving each node). After the AAID algorithm applied, the action matrix pairs are:

Node 0: (initial, root node)

A[0][0] matrix (i.e., the elements of A[0][0][0][0] through A[0O][0][2][2]):

1 0 0

0 1 0

0 0 1

B[0][0] vector (i.e., the elements of B[0][0][0][0] through B[0][0][0][2]

0

0

0

Node 1:

A[1][0] matrix:

0 0 0

0 1 0

0 0 1

```
B[1][0] vector:
        10
        11
        0
A[1][1] matrix:
        0 0 0
        0 1 0
        0 0 1
B[1][1] vector:
          20
          21
          0
Node 2:
A[2][0] matrix:
        0 0 0
        0 1 0
        0 0 0
B[2][0] vector:
          10
          11
          1
A[2][1] matrix:
        0 0 0
        0 1 0
        0 0 0
B[2][1] vector:
        10
        11
        -1
A[2][2] matrix:
        0 0 0
        0 1 0
        0 0 0
B[2][2] vector:
          20
          21
          1
A[2][3] matrix:
        0 0 0
        0 1 0
        0 0 0
B[2][3] vector:
        20
        21
        -1

Node 3:
A[3][0] matrix:
        0 1 0
        0 1 0
        0 0 0
B[3][0] vector:
        22
        11
        1
A[3][1] matrix:
        0 1 0
        0 1 0
        0 0 0
B[3][1] vector:
        20
        11
        -1
A[3][2] matrix:
        0 1 0
        0 1 0
        0 0 0
B[3][2] vector:
          42
          21
           1
A[3][3] matrix:
        0 1 0
        0 1 0
        0 0 0
B[3][3] vector:
        40
        21
        -1
```

A[3][4] matrix is the same as A[2][0] matrix.

B[3][4] vector is the same as B[2][0] vector.

A[3][5] matrix is the same as A[2][1] matrix.

B[3][5] vector is the same as B[2][1] vector.

A[3][6] matrix is the same as A[2][2] matrix.

B[3][6] vector is the same as B[2][2] vector.

A[3][7] matrix is the same as A[2][3] matrix.

B[3][7] vector is the same as B[2][3] vector.

In the outgoing edges of Node 2 AAI are detected.

B6. AAID Algorithm Advantages

The advantage of the AAID algorithm will be most visible when it is combined with the AAI removal algorithm in the next stage of the project. By those algorithms, the AAI's are eliminated without unnecessarily creating a large number of nodes in the EFSM graph, which is essential for generating a near-minimum number of test cases for the VHDL specifications.

Although AAID uses an approach similar to the symbolic execution method used in software engineering, it is more powerful than the symbolic execution. Symbolic execution is based on the data flow graph model of a specification which lacks the details of conditions that are logically AND'ed/OR'ed. AAID algorithm uses a directed graph based on the EFSM where conditions using logical AND/OR are represented by separate edges. This graph will yield test cases with better coverage than the ones based on the data flow graph oriented methods.

B7. AAID Algorithm Limitations

Since the conditions are not considered at this point of the project, there is no criteria for stoppage when a previously visited node is reached. For now, it is assumed that each node will be visited at most 3 times (an arbitrary number). This restriction will be removed when AAID will be expanded to include the conditions. Therefore, the traversal does not continue when either a node is visited 3 times or the root node is reached. When there are no nodes to visit, AAID terminates.

For simplicity, all variables are assumed to be homogenized for all actions (i.e., if a variable is not used in a action, its coefficient is 0). This assumption will be easily handled when VHDL specification is translated into an EFSM graph. The actions and the conditions in the specification are assumed to be in linear form.

C. Metrics

C 1. Performance Parameters

The run time for the AAID is polynomial with respect to the graph size of the EFSM model, as can be seen from the high level algorithm description.

The total number of action update matrices is directly proportional to the number of paths in the EFSM graph. For the worst case, the number of matrices can be infeasibly large. However, for the average case (i.e., in the specifications where variables do not get updated by many other variables hence creating dependencies among large number of variables), the number of matrices and the size of each matrix is expected to be feasible.

4.2 Algorithm for A-A Inconsistencies Removal (AAR)

A. Problem Statement

Al. Action-to-Action Inconsistency Definition:

In an EFSM representing a VHDL specification, consider an action that modifies a variable (say vl) by using other variable(s) (say v2) and/or constants. If v2 has multiple values due to previous modifications in the specification, an action-to-action inconsistency (AAI) occurs. In general, whenever a variable used in an action has more than one value (due to previous modifications) an inconsistency is present since it is not clear which value of the variable wfll be used at that point.

A2. Examples for the general case:

Consider the following action where x is assigned two different constant values based on condl:

```
if (condl = 1) then

        x = 10;

else
        x = 20;

endif
```

If x is used later in the specification, an AAI will occur since x can have two different values depending on condl value.

Consider the following action where y is updated differently based on cond I:

```
if(condl)then

      y = y + 11;
else

      y = y + 21;

endif
```

Regardless of the initial value of y, an AAI will occur if y is used later in the specification since y will have two different values.

A3. Examples for the VHDL case:

The following portion of a VHDL specification contains the AAI based on assigning constants to variables or updating variables in actions differently:

```
if (cond1 = 1) then
        x = 10;
        y = y + 11;
else
         x = 20;
         y = y + 21;
endif
if (cond2 = 1) then
        z = 1;
else
         z = -1;
endif
if (cond3 = 1)
         x = x + y + z;
endif
```

In the above example, the if statement with cond3 uses x, y, and z in its action where each has different values based on which conditions are satisfied above (condl and cond2). If, for example, (condl = 1) and (cond2 = 1) were satisfied above, the new value for x in if (cond3=1) will be:

$$x = (10) + (y+ 11) + (1) = y + 22$$

On the other hand, if (cond1 = 0) and (cond2 = 0) above, the new x value in if (cond3 = 1) will be:

$$x = (20) + (y + 21) + (-1) = y + 40$$

Based on the initial value of y, the following values are possible for x after if(cond3=1)else is executed:

x = y + 40	(condl=l,cond2=1,cond3=1)
x = y + 22	(condl--O,cond2=0,cond3=1)
x = y + 20	(condl=l, cond2=0, cond3=1)
x = y + 42	(condl=0, cond2=1, cond3=1)
x = 10	(condl=l, cond2=1, cond3=0)
x = 20	(condl=0, cond2=0, cond3=0)
x = 10	(condl=l, cond2=0, cond3=0)
x = 20	(condl=O, cond2=1, cond3=0)

Similarly, y and z have 2 possible values each, based on the values of condl and cond2.

Since only the AAIs are considered at this point in the project, it is assumed that the conditions of cond1, cond2 and cond3 are conflict-free and that they are using variables other than x, y, and z.

A4. Action-to-Action Inconsistency Removal (AAIR) Method:

At this stage of the project, only the inconsistencies among actions of a directed graph representation of an EFSM are considered. The directed graph that is used in the AAIR method is based on the traditional data flow graph (DFG); since this directed graph represents the logical AND/OR conditions as different edges, it is more powerfid than the traditional DFG model.

The lack of condition variables in this analysis creates a high degree of commonality between the method for the action-to-action inconsistency detection (AAID), and the AAIR method discussed here. In fact, the method presented for the AAID will also be sufficient to remove the AAI.

To prevent unnecessary creation of nodes in the directed graph,splitting of nodes are not needed when an AAI is detected.

As in the AAID method, the directed graph representation of the EFSM is traversed in a breadth-first manner. The action statements are examined and the variable updates are accumulated. When different values for a given variable on the right-hand side of an action are observed, an AAI is detected. For each AAI,A different update matrix pair is created. The effect of creating such distinct action update pairs, when the conditions are not considered in the inconsistencies, is equivalent to the removal of these AAI from the actions of the graph edge.

B. Proposed Solution

B I. Solution Formulation

Inconsistencies among the variables used in the left hand sides of the actions (i.e., the variables that are modified within the specification) of a VHDL specification statements are considered. As mentioned above, it is assumed that the variables that are updated within the specification are not used in the conditions, which would have created action-to-condition inconsistencies.

The AAIR method will utilize the algorithms provided for the AAID method. To prevent the creation of unnecessary nodes in the directed graph representation of a VHDL specification, the nodes are not split due to the detected AAI. Instead, different action update matrices will be accumulated for each node and node splitting will be presented when action-to-condition inconsistencies are handled.

The AAIR algorithms use the same data structures as the AAID algorithms. Each node is associated with a pair of matrices:

A[N][Q][V][V]: action update matrices representing all the modifications applied to every variable. The upper bounds for N, Q, and V are:

N: number of nodes in the graph,

Q: number of valid paths accumulated up to a given node,

V: number of variables in the graph,

B[N] [Q] [V] [I]: scalar values for the modified variables.

The A matrix for the root node (i.e., the initial node of the graph), A[root][0][V][V], is initialized to the identity matrix of I[V][V]. Elements of B[root][0][V][1] are initialized to 0.

As in the condition-to-condition inconsistency detection and removal algorithms, for simplicity, all variables are assumed to be homogenized for the graph for all actions (i.e., if a variable is not used in a condition, its coefficient is 0).

The actions and the conditions in the graph are assumed to be in linear form.

In a given node n_i in the graph, an outgoing edge e_i may have several actions. Each action is in the form of:

x_r = D[I] [V] (matrix mult) X[V] [I] + s_r where the V-element vector D represents the coefficients of the variables which modify x_r, X is the vector of variables, and s_r is the scalar used in this action.

There is a harness called BFM_DRIVER, which traverses the graph in a breadth-first manner. During the traversal, for each node, the action-to-action inconsistencies are checked by DET_-ACT_ACT. UPDATE_ACT modifies the variables based on the actions of each edge while traversing the graph. The updated actions are kept in the A and B matrix pairs associated with each node.

Since the conditions are not considered in this path, there is no criteria for stoppage when a previously visited node is reached. For now, it is assumed that each node will be visited at most 3 times (an arbitrary number). This restriction will be removed when this algorithm will be expanded to include the conditions. Therefore, the traversal does not continue when either a node is visited 3 times or the root node is reached. When there are no nodes to visit, the algorithm terminates.

B2. AAIR Algorithm

Similar to the case of AAID, the graph is traversed in the breadth-first-manner. When each node is visited, all actions associated with the outgoing edges of the node are updated by checking the right hand sides of the updated variables. If a variable x is updated by another variable y, where y is modified differently in two or more parallel edges above, an AAI is detected (x and y may or may not be the same variable). This AAI is removed from the graph by creating a different acfion matrix pair which will be associated with the ending node of that edge.

A high-level description of the AAIR algorithm is as follows:

```
 in a breadth-first manner visit each node i:
 /* check inconsistencies for edges leaving node i */
for (each ending node j from node i) {

      for (each outgoing edge from node i to a node j) {

              for (each action on an edge) {

                     check for AAI by comparing the accumulated matrix pairs for node i

                     with the actions on the edge. If the edge actions are using

                     variable updated differently above, AAI is detected.

                     }
}
for (each parallel edge from node i to node j) {
        Based on the actions on the edge, update the action matrix pairs so that they reflect the
        latest action modifications on the variables.  By providing different action matrix pairs
        for each different modification of variables, AAI are removed from the graph.
        }
}
```

The termination criteria for the AAIR method is the same as in the AAID method. The breadth-first traversal does not continue with a node which is either the root node or a node that is already visited 3 times (an arbitrary number). This limitation is needed since the condition-to-condition inconsistencies are not considered in the graph (which will cause infinite-loops since without the conditions the loops do not end). When AAID and AAIR are combined with the methods that consider conditions, this limitation will be removed (See also Section B8 on the limitations of the AAIR algorithm).

B3. AAIR Algorithm compared to methods in literature

The methods for AAID/AAIR use an approach similar to the symbolic execution method defined in the field of software engineering. The symbolic execution is designed for the graph representation of a program whereas AAID/AAIR methods are defined for the EFSM model. In general, EFSM model contains more relevant information from the test generation point of view than the data flow graph. This extra information of EFSM is especially important when eliminating the inconsistencies.

B4. AAIR Applied to the General Case

The application of AAIR for the general case is the same as in the VHDL case (See Section B5).

B5. AAIR Applied to the VHDL Case

In general, the actions in each edge will be in the form of.

x   a00 a01 a02 x   b00

y = al0 al1I al2 x y + b0l

z   a20 a2l a22   z   b02

For example: y = y + 11I will be:

$$\begin{matrix} x \\ y \\ z \end{matrix} = \begin{matrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{matrix} \times \begin{matrix} x \\ y \\ z \end{matrix} + \begin{matrix} 0 \\ 11 \\ 0 \end{matrix}$$

The EFSM graph of the VHDL specification from Section A3 has 4 nodes (0 through 3) and 8 edges (2 edges leaving each node). After the AAID algorithm applied, the action matrix pairs are:

Node 0: (initial, root node)

A[0][0] matrix (i.e., the elements of A[0][0][0][0]

through A[0][0][2][2]):

1 0 0

0 1 0

0 0 1

B[0][0] vector (i.e., the elements of B[0][0][0][0] through B[0][0][0][2]

0

0

0

Node 1:

A[1][0] matrix:

0 0 0

0 1 0

0 0 1

B[1][0] vector:

10

11

0

A[1] [1I] matrix:

0 0 0

0 1 0

```
            0 0 1
B[l][1] vector:
            20
            21
            0

Node 2:
A[2][0] matrix:
            0 0 0
            0 1 0
            0 0 0
B[2][0] vector:
            10
            11
            1
A[2][1] matrix:
            0 0 0
            0 1 0
            0 0 0
B[2][1] vector:
            10
            11
            -1
A[2][2] matrix:
       0 0 0
       0 1 0
       0 0 0
B[2][2] vector:
            20
            21
```

```
          1
A[2][3] matrix:
      0 0 0
      0 1 0
      0 0 0
B[2][3] vector:
      20
      21
      -1

Node 3:
A[3][0] matrix:
      0 1 0
      0 1 0
      0 0 0
B[3][0] vector:
          22
          1 1
A[3][1] matrix:
          0 1 0
          0 1 0
          0 0 0
B[3][1] vector:
          20
          11
          -1
A[3][2] matrix:
      0 1 0
      0 1 0
      0 0 0
```

B[3][2] vector:

42

21

1

A[3][3] matrix:

0 1 0

0 1 0

0 0 0

B[3][3] vector:

40

21

-1

A[3][4] matrix is the same as A[2][0] matrix.

B[3][4] vector is the same as B[2][0] vector.

A[3][5] matrix is the same as A[2][1] matrix.

B[3][5] vector is the same as B[2][1] vector.

A[3][6] matrix is the same as A[2][2] matrix.

B[3][6] vector is the same as B[2][2] vector.

A[3][7] matrix is the same as A[2][3] matrix.

B[3][7] vector is the same as B[2][3] vector.

In the outgoing edges of Node 2, AAIs are detected. Representing the AAIs as different action matrix pairs is equivalent to removing them from the graph.

Within the scope of this example, consider a case where there were more nodes after these 3 nodes in the specification which were using the variables x, y, and/or z. Creating different action update matrices represent all of the different ways that x, y, and z were updated by the actions of the edges leaving nodes 0, 1, and 2. This implies that there would be no AAI when x, y, and/or z were used later in the graph.

B7. AAIR Algorithm Advantages

The main advantage of the AAID/AAIR algorithms is the avoidance of unnecessary node splits when eliminating the inconsistencies among the actions and conditions. Note that the number of nodes in the graph after the inconsistencies are eliminated is directly proportional to the number of tests for a given VHDL specification.

Although the approach is similar to the symbolic execution method of software engineering, the AAID/AAIR methods are more powerful. The symbolic execution is based on the traditional DFG model of a specification which lacks the details of conditions that are logically AND'ed/OR'ed. The directed graph used in the AAID/AAIR algorithms are also based on the DFG however they represent the logical AND/OR conditions by separate edges. This graph will yield test cases with better coverage than the ones based on the traditional DFG oriented methods.

B8. AAIR Algorithm Limitations

Since the conditions are not considered at this point of the project, there is no criteria for stoppage when a previously visited node is reached. For now, it is assumed that each node will be visited at most 3 times (an arbitrary number). This restriction will be removed when AAIR will be expanded to include the conditions. Therefore, the traversal does not continue when either a node is visited 3 times or the root node is reached. When there are no nodes to visit, AAIR terminates.

As in the condition-to-condition inconsistency detection and removal algorithms, for simplicity, all variables are assumed to be homogenized for the graph for all actions (i.e., if a variable is not used in a condition, its coefficient is 0). This assumption will be easily incorporated into the graph representation during the VHDL specification translation into an EFSM graph.

The actions and the conditions in the specification are assumed to be in linear form.
For the first version of AAID/AAIR, the actions using logical operations (e.g. x = y AND z), the array elements, the use of types (as opposed to the direct forms such as integer, real, and Boolean variables) are excluded. These simplifications do not effect the algorithms but only the translation from VHDL to EFSM.

C. Metrics

C I. Performance Parameters

The run time for the AAID/AAIR are polynomial with respect to the graph size of the EFSM model, as can be seen from the high level algorithm description.

The total number of action update matrices is directly proportional to the number of paths in the EFSM graph. For the worst case, the number of matrices can be unfeasibly large. However, for the average case (i.e., in the specifications where variables do not get updated by many other variables hence creating dependencies among large number of variables), the number of matrices and the size of each matrix is expected to be feasible.

C2. Metrics for the ALU and the Microcontroller

At this point of the project, it is too early to determine. Meaningful metrics will be developed as the algorithm development progresses.

5.0 AADR Pseudo-code

```
DFM_DRIVER(root)
{
cur_nodes[O] = root;
 cur_nodes[ I] = null;
 next_nodes[O] = null;

/* for all nodes count is initialized to 0 */
node-visit-Count[*] = 0;
DFM_DONE = 0;
while (DFM_DONE == 0) {
for (each node i in cur-nodes[*]) {
   /* check inconsistencies for edges leaving node i */
   for (each ending node j from node i) {
      DET_ACT_ACT(node i, node j, A_2_A);
      if (A_2_A == true) {
         /* an action-to-action inconsistency is detected on the edges from node i to j;
         creation of different action update matrices will eliminate the action to action
         inconsistencies. */
   }
else  {}
/* update actions on the edges from node i to node j based on the action updates of node i so
far*/
for (each parallel edge m from node i to node j) {
   /* update the actions one by one in the order they are in the graph edge from node i to j */
   for (each action k in edge m from node i to node j) {
      UPDATE_ACT(node i, node j, action k);
      }
   }
/* update the list of next nodes to visit */
```

```
     if (node_visit_count[j] <= 3 && node j != root) {
          if (node j is not in next_nodes[*]) {
                next-nodes[*] +=node j;
                node_visit_count[j]++;
                }
           else {}
            }
else {}
} /* end j loop */
      } /* end i loop */
   /* copy the list of next nodes into the current nodes */
   cur_nodes[*] = next_nodes[*];
   next-nodes[0] = null;
   if (cur_nodes[0] == null) {
       DFM_DONE = 1;
   else {}
   } /* end DFM loop */
} /* end of DFM_DRIVER */
```

The pseudo code for DET_ACT_ACT:

```
DET_ACT_ACT(node i, node j, A_2_A) {
{
```

Suppose the matrix pairs for node i are A[i][q][V][V] and B[i] [q] [V] [I] where i represents the node i, q is the number of different of matrix pairs associated with node i, V is the number of variables. An action which modifies the variable x-r in an edge from node i to node j is in the form of:

X_r = (D[l][V] (matrix mult) X[V][1]) + s_r

To simplify the notation, let us use a[V][V] and b[V][1] to denote one of the q matrix pairs associated with node i. The returned parameter A 2 A is set to true is an action-toaction inconsistency is detected. This returned variable will not be used in the current version of the AAIR method. However, it is expected to be used when combining actions with conditions at the later state of the project.

```
for (each outgoing edge p from node i to node j) {
  for (each action k on edge p) {
     /* check for action-to-action inconsistencies */
     for (each nonzero coefficient d in D[ I] [V]) {
        /* copy the dth row of a[V][V] into temp_yow[l][V] */
          temp_row[1][V] = a[d][V] for (each one of q matrix pairs of node i) {
             compare dth row of a A[i][q][V]
             with temp row[1][V], element by element; if any element is different, an action-
             to-action inconsistency is detected; set A_2_A = true;
             }  /* end of q loop */
          } /* end of d loop */
       } /* end of k loop */
    } /* end of p loop */
 } /* end of DET_ACT_ACT
 UPDATE_ACT(node i, node j, action k):
  {
```

Suppose the matrix pairs for node i are A[i][q][V][V] and B[i][q][V][1] where i represents the node i, q is the number of different of matrix pairs associated with node i, V is the number of variables. An action which modifies the variable x_r in an edge from node i to node j is in the form of.

```
  x_r = (D[ I ] [V] (matrix mult) X[V] [ I ]) + s_r
```

To simplify the notation, let us use a[V][V] and b[V][1] to denote one of the q matrix pairs associated with node i.

```
  /* compute the V-element vector temp_row and the scalar temp_const */
  temp_row[ I ] [V] = D[ I ] [V] (matrix mult) a[V] [V];
  temp_const = (D[ I ] [V] (matrix mult) B [V] [ I ]) +  s_r;
  /* replace the rth row of a[V] [V] by temp_row[ I ] [V] */
  a[r] [V] = temp_row[ I ] [V];
  /* replace the rth element of b[V][1] with temp_const */
  b[r][1] = temp_const;
```

```
} /* end of UPDATE_ACT
```

6.0 Implementation of the Inconsistencies Detection and Removal Algorithm

In the first stage of the conversion, we have used the V2SIF package developed by the University of Cincinnati. V2SEF accepts VHDL specifications as input. The Figure 2 shows how the implementation progresses after that:

First the intermediate form file (.sif) is converted to a data flow graph form (.dfg) . To convert the data flow graph to an extended finite state machine (EFSM) graph a program called dfg2efsm was developed . Essentially, this program "demasks " those variables that have been masked by the sif format. Finally efsm2fsm detects and removes the condition-to- condition inconsistencies and hence producing the final FSM form.

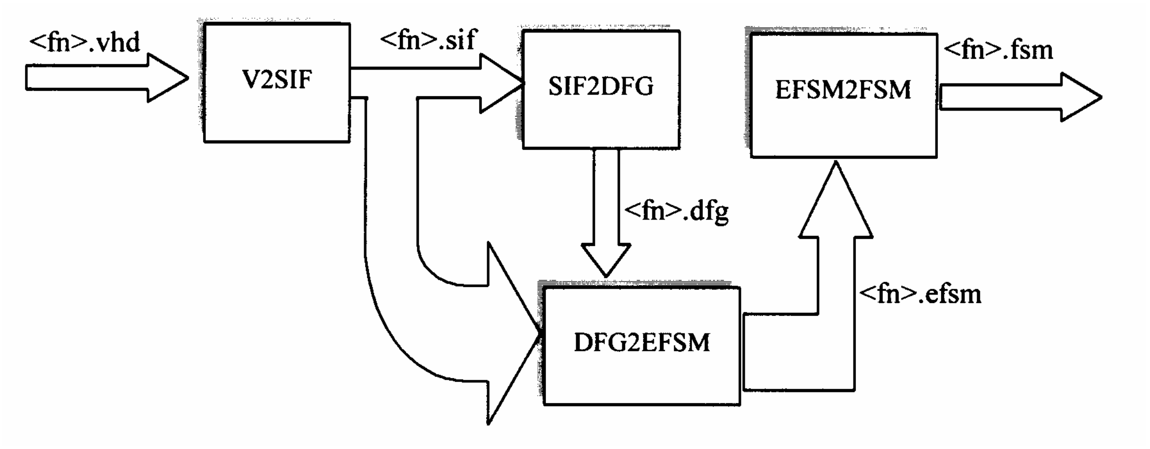


Figure 2: Implementation Process

7.0 Summary

In this report an algorithm was developed to detect the inconsistencies among the conditional and action statements of the EFSM graph. A second algorithm removed the inconsistencies from the EFSM graph by splitting the vertices. In the implementation phase the conversion of the VHDL specification into an EFSM directed graph was performed using the V2SEF tool. The EFSM without the inconsistencies is the FSM model of the original VHDL specification. It is perhaps important to mention at this point that the proposed method of test generation will be limited to a subset of VHDL, since considering the full capabilities of VHDL will be too difficult within the scope of this project. In fact, the solution for the general case is an open problem.

8.0 References

[1] V.D. Agrawal and S.C. Seth, "Tutorial: test generation for VLSI chips," Computer Society Press, 1988.
[2] H.B. Min, H. A. Luh and W.A. Rogers, "Hierarchical test pattem generation: a cost model and implementation," IEEE Trans. on CAD, Vol 12, pp. 1029-1039, Jul 1993.
[3] K.T. Chen and A.S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," Proc. Design Automation Conf, pp. 86-91, 1993.
[4] B.S. Bosik and M.U. Uyar, "FSM based formal methods in protocol conformance testing: from theory to implementation," Computer Networks and 1[5] P.D. Coward, "Symbolic execution and testing," Information and Software Technology, Vol. 33, pp. 53-64, Jan. 1991.
[6] 0. Coudert, C. Berthet and J. Madre, "Verification of sequential machines using boolean functional vectors," Proc. IMEC-IFIP Int'l. Workshop on Applied Formal Methods for Correct VLSI Design, pp. 111-128,1989.
[7] A. Mujumdar, R. Jain and K. Saluja, "Incorporating testability considerations in high-level synthesis, "Jour. Electronic Testing: Theory and Applications, Vol. *5*, pp. 43-559 1994.
[8] M. Karam and G. Saucier, "Functional versus random test generation for sequential circuits," Jour. Electronic Testing: Theory and Applications, Vol. 4, pp. 33419 1993.
[9] P. Anirudhan and P. Menon, "Symbolic test generation for hierarchically modeled digital systems," Proc. Int'l. Test Conf., pp. 461-469, 1989.
[10] J. Lee and J. Patel, "Architectural level test generation for microprocessors," IEEE Trans. on CAD, Vol 13, pp. 1288-1300, Oct. 1994.
[11] S. Bhattacharya, F. Brglez and M. Pilsl, "Synthesis for testability from behavioral specifications: benchmarking the scheduling strategies," MCNC Techn. Report June 1993.
[12] M. U. Uyar and A. Duale, "Inconsistencies in EFSM Models for VHDL Specifications, " Proc. 1st Advanced Telecommunications/Information Distribution Research Program (ATRIP) Conference, College Park, MD, Jan 1997.
[13] R.J. Linn and M.U. Uyar, "Tutorial: conformance testing methodologies and architectures for OSI protocols," IEEE Computer Society Press, 1994.
[14] A.T. Dahbura, M.U. Uyar and C.W. Yao, "An optimal test sequence for the JTAG/IEEE PI 149.1 test access port controller," Proc. Int'l. Test Conf., pp. 55-62, 1989.
[15] K.K. Sabnani and A.T. Dahbura, "A protocol test generation procedure," Computer Networks and ISDN Systems, Vol 15, No. 4. pp. 285-297, 1988.